7. Dynamic programming

Dynamic programming is a technique for designing algorithms to solve problems by combining solutions to subproblems (just like divide-and-conquer). In some cases, subproblems overlap with each other, and because of this overlapping among subproblems, using simple recursive algorithms to solve the problem may result in solving the same subproblem several times! Dynamic programming avoids this by solving each subproblem only once and recording the result in a table. While divide-and-conquer is applicable to solve problems where subproblems are independent, dynamic programming is applicable to solve problems where subproblems overlap.

This can be illustrated using the problem of computing Fibonacci numbers \( (F_n) \), which can be defined as follows.

\[
F_0 = 0, \quad F_1 = 1, \quad \text{and} \quad F_n = F_{n-1} + F_{n-2} \quad \text{for} \quad n \geq 2.
\]

A simple recursive algorithm to compute \( F_n \) could be described as follows:

```plaintext
Fibonacci (n)
if n < 2 then
    return n
else
    return Fibonacci (n - 1) + Fibonacci (n - 2);
end
```

However, by tracing the above recursive algorithm, one can easily find that many subproblems are solved again and again. For example, let’s consider computing \( F_5 \) using the
7. Dynamic programming

recursive algorithm. There will be one call for \( n = 5 \) and \( n = 4 \); two calls for \( n = 3 \); three calls for \( n = 2 \); five calls for \( n = 1 \); and three calls for \( n = 0 \). It would be better if we solve the problem by one call for each value of \( n \). Now let’s look at a dynamic programming version of the this algorithm.

\[
\text{Fibonacci} \ (n) \\
A[0]:=0; \\
A[1]:=1; \\
\text{for } i = 2 \text{ to } n \text{ do } \\
\quad A[i]:=A[i-1]+A[i-2]; \\
\text{end} \\
\text{return } A[n];
\]

Note that while the simple recursive algorithm and divide-and-conquer algorithms in general take a top-down direction, dynamic programming algorithms usually take a bottom-up direction.

Example: computing a binomial coefficient \( C(n, k) \).
\( C(n, k) \), or \( \binom{n}{k} \), can be interpreted as the number of different \( k \)-element combinations out of \( n \) elements.
\( C(n, 0) = C(n, n) = 1 \) and \( C(n, k) = C(n - 1, k) + C(n - 1, k - 1) \), for \( n > k > 0 \).

\[
\text{Binomial} \ (n, k) \\
\text{for } i = 0 \text{ to } n \text{ do } \\
\quad \text{for } j = 0 \text{ to } \text{MIN}(i, k) \text{ do } \\
\quad \quad \text{if } j = 0 \text{ or } j = i \text{ then } \\
\quad \quad \quad C[i, j]:=1; \\
\quad \quad \text{else } \\
\quad \quad \quad C[i, j]:=C[i-1, j-1]+C[i-1, j]; \\
\quad \text{end} \\
\text{end} \\
\text{return } C[n, k];
\]
7. Dynamic programming

Time complexity

The basic operation in this algorithm is the addition. We can compute the step count of this operation as follows.

\[ A[n, k] = \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n} \sum_{j=1}^{k} 1 = \sum_{i=1}^{k} (i - 1) + \sum_{i=k+1}^{n} k = \frac{(k-1)k}{2} + k(n - k) \in \Theta(nk). \]